Left Resursion

Left Recursive Grammar

A grammar is left recursive if it has a non terminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α. This process is also known as *immediate left recursive* grammar.

Problem

- Execute the production of a non-terminal symbol without checking the terminal symbol which prosuce infinite loop or iteration.
- As it executes the production of a non-terminal symbol, so, this is the way to increse possibility to make a grammar ambiguous.
- A left recursive grammar make the iteration with $\beta\alpha^*$, if the grammar is $A \rightarrow A\alpha \mid \beta$

Ambiguous Grammar

A grammar is called **ambiguous** if we find -

- more than one left most derivation, or
- more than one rightmost derivation, or
- more than one parse tree

Example:

We have a grammar:

$E \rightarrow E + E \mid E * E \mid id$

and the corresponding string is id + id * id, we will get two distinct left most derivation.

Solving issue

Care about the precedence of + and *, where we have to treat operator * as higher precedence than +, correspndiong to fact that we would normally evaluate an expression like id+id*id as id+(id*id) rather than (id+id)*id.

Elimination of Left Recursive Grammar

Concept

We have to the eliminate the left recursion without changing the language $\beta\alpha^*$

Formula/Grammar to eliminate the Left Recursion

For the grammar $A \rightarrow A\alpha \mid \beta$.

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Example:

Eliminate the left recursion for the following grammar

$S \rightarrow S\ 0\ S\ 1\ S \mid 01$

Solution:

$S \rightarrow 01\ S'$

$S' \rightarrow 0\ S\ 1\ S\ S' \mid \epsilon$

Assignment 04: Parse tree and Ambiguity

There is given some context free grammar with a string. You have to do the following operations for each grammar and string:

I. You have to find the parse tree for each grammar corresponds to the string.

II. Justify you answer whether each grammar is ambiguous or unambiguous.

1. $S \rightarrow S\ S\ + \mid S\ S\ * \mid a$ with the string aa+a*.
2. $S \rightarrow 0\ S\ 1 \mid 0\ 1$ with the string 000111.
3. $S \rightarrow +\ S\ S \mid *\ S\ S \mid a$ with the string +*aaa.
4. $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$ with the string (()()).
5. $S \rightarrow S + S \mid S\ S \mid (\ S\ ) \mid S\ * \mid a$ with the string (a+a)*a.
6. $S \rightarrow (\ L\ ) \mid a$ and $L \rightarrow L, S \mid S$ with the string ((a,a),a,(a)).
7. $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$ with the string aabbab.

...ignment 05: Left Recursion

Justify the following grammars and eliminate left recursion if it has.

1. L → L , S | S
2. E → E + T | T
3. E → E + T | T
   T → T * F | F
   F → (E) | id
4. E → T c | a
   T → F * T | E y d | ϵ
   F → T | id
5. E → E + E | E * E | id
6. A → A B d | A a | a
   B → B e | b

Assignment 05: Left Recursion

7. S → A
   A → A d | A e | a B | a c
   B → b B c | f
8. A → A A α | β
9. A → B a | A a | c
   B → B b | A b | d
10. X → X S b | S a | b
    S → S b | X a | a
11. A → B x y | x
    B → C D
    C → A | c
    D → d

Eliminate the Left Recursion

Eliminate the Left Recursion for multiple productions

Now, whatif you have more one or multiple productions as following grammar:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

Modified Grammar

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Example

Eliminate the left recursion for the following grammar

E → E + T | E * T | a | b

Solution

E → aE′ | bE′
E′ → +TE′ | *TE′ | ϵ

Non immediate left recursion

Consider the following grammar:

S → A a | b
A → A c | S d | ϵ

The non terminal S is left recursive because S ⇒ Aa ⇒ Sda, but this is not immediately left. On the other hand we can write A → A c | A a d | b d | ϵ.

Eliminate the left recursion

S → A a | b
A → bdA′ | ϵA′
A′ → cA′ | adA′ | ϵ

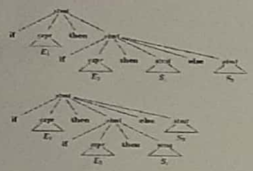
factoring not removes ambiguity

dangling-else

Consider the following grammar, which is called "dangling-else".

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | a
```

We can write the grammar as
S → i E t S | i E t S e S | a. Here, `stmt` means S, `if` means i, `expr` means E, and `then` means t.
So, this grammar is ambigous for the string
`if` E_1 `then if` E_2 `then` S_1 `else` S_2
as it produced two different perse trees.

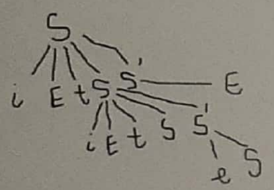


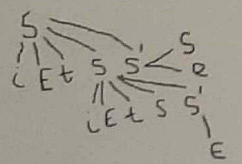
Left factoring not removes ambiguity

Left factoring the "dangling-else" grammar

S → i E t S S′ | a
S′ → ϵ | e S

Now, if we derive the string i E t i E t S e S, then we will get again two different parse trees.





In that case, we can say that eliminating the non-determinism by left factoring does not remove the ambiguity of a grammar.

Non-determinism

Introduction

Non-determinism means you have many options on a single symbol. Consider the following grammar: A → $\alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$. From this grammar you have to derive the string $\alpha\beta_3$.
In order to derive the string from the grammar you may have needed the **backtracking**. In addition, that backtrack is happened because of **common prefixes** and this is the concept of non-determinism.

Left Factoring

If you have a non-deterministic grammar as following:
A → $\alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$ | ... | γ
So, you have to postpone the decision making problem until find to β_3.

Left Factoring

The equivalent deterministic grammar is:

A → αA′ | γ
A′ → β_1 | β_2 | β_3

Example

Do left factoring the following grammar:

S → b S S a a S | b S S a S b | b S b | a

Solution

S → b S S′ | a
S′ → S a a S | S a S b | b
S′ → S a S″ | b
S″ → a S | S b

oduction to FIRST and FOLLOW

FIRST

FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \Rightarrow \epsilon$, then ϵ is also in FIRST(α).

Example: Consider the following grammar:

```
S → a A B C D
A → b
B → c
C → d
D → e
```

```
FIRST(S) = FIRST(a) = {a}
FIRST(A) = FIRST(b) = {b}
FIRST(B) = FIRST(c) = {c}
FIRST(C) = FIRST(d) = {d}
FIRST(D) = FIRST(e) = {e}
```

Introduction to FIRST and FOLLOW

Example: Consider the following grammar:

```
S → A B C D
A → b | ε
B → D c | ε
C → d
D → e | f | ε
```

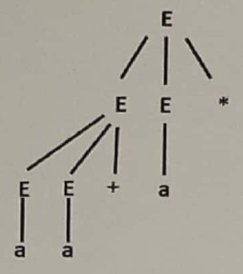
Solution:

```
FIRST(S) = FIRST(A) = {b, ε} = {b}
= FIRST(B) = {e, f, c, ε} = {e, f, c}
= FIRST(C) = {d} = {b, e, f, c, d}
FIRST(A) = FIRST(b) = {b} = FIRST(ε) = {ε}
= {b, ε}
FIRST(B) = FIRST(D) = {e, f, ε} = {e, f}
= FIRST(c) = {c} = FIRST(ε) = {ε} = {e, f, c, ε}

FIRST(C) = {d}
FIRST(D) = {e, f, ε}
```

Assignment 06: Left factoring

1. Do left factoring for the following grammar:
   S→a S S b S | a S a S b | a b b | a
2. There is given a perse tree. Find the corresponding grammar and make deterministic if it is not.



3. Why a non-deterministic grammar can not work to design a compiler?

Assignment 06: Left Factoring

4. Describe the following terms:
   i. Ambuguity
   ii. Left recursion
   iii. Deterministic and non-deterministic grammar
   iv. Left factoring
5. The determinism of a grammar does not remove the ambiguity - describe it.
6. Do left factoring for the following grammars:
   i. A → a A B | a B c | a A c
   ii. S → a | a b | a b c | a b c d
   iii. S → a A d | a B
   A → a | a b
   B → c c d | d d c